CCDAO Crypto Collective Yield Optimizer Security Review

This document is meant to represent a security review performed on the codebase of `raikkao` meant to implement the CCYO, a yield-optimizer vault implementation users can deposit to to have their funds automatically deployed in various yield-optimized strategies and be rewarded with a native governance token.

General Remarks

This section will contain general remarks on the codebase that cannot be applied to a single finding and are meant to reference the overall state of the codebase.

Dependencies

The code does not make use of any package level imports and instead contains local copies of the contracts it utilizes. This is ill-advised due to hotfixes being regularly released for old package versions as well as for the legibility of the codebase itself as project-specifier libraries and i.e. OpenZeppelin libraries are mixed together in a single folder. It is strongly recommended to simply use an `@openzeppelin/contracts` dependency instead with a proper `package.json` setup etc.

Additionally, it appears that `Math` and `OptimizedMath` are equivalent and it appears as if some contracts have multiple top-level declarations (i.e. `Timelock` has `SafeMath` internally declared etc.) which can cause ambiguities in static analysis tools as well as the compiler itself. In general, improper inheritence structures are observed across the codebase (i.e. `CryptoCollectiveCoin` inheriting both `ERC20` and `ERC20Mintable` and containing a top-level declaration of `CCC` instead of `CryptoCollectiveCoin`) and we advise them to be streamlined across the codebase.

Compilation

The codebase of the project contains an open-ended `pragma` version specifier (`^0.6.12`) which renders the detection of compiler-specific vulnerabilities difficult to assess. Granted, the version specified only allows the compiler version of `0.6.12` to be utilized it is still strongly recommended to change all specifiers to a locked pragma version (`=0.6.12`) to avoid any ambiguities due to potential future hotfixes etc.

Compilation was not possible due to incorrect `import` paths within all contracts in the `contracts/strategies/curve` folder path. Once a folder-up operator (`..`) was introduced to each `import` statement compilation was successful.

Linting

The codebase appears to have an absence of a linting guideline. To amend this, we strongly recommend the usage of `prettier-plugin-solidity` either independently or via the `solidity` plugin of Juan Blanco on VS Code. To properly apply it, instructions are provided in this section of the `prettier-plugin-solidity` plugin as VS Code does not work out of the box with it.

Manual Review Findings

These consist of manual review findings that would affect the security of the project from either a tangible perspective (i.e. exploitable attack vector) or a standards perspective (i.e. inapplicacy of best practices). The severity of these findings ranges from "minor" to "major" depending on a personal assessment of likelihood, impact, and privilege required. These are suffixed with the `M` identifier.

Static Analysis Findings

These consist of static analysis findings that have varying degrees of impact but are detected with the assistance of static analysis tools rather than manual review. They are explicitly labelled for the sake of transparency and have been manually filtered to ensure no false positives are relayed to CCDAO. These are suffixed with the `S` identifier.

Code Style Findings

These consist of code style findings that are all of `informational` severity and do not affect the actual security of the project. Instead, they refer to best code style practices as well as gas optimizations that can be applied to the codebase. These are suffixed with the `C` identifier.

Whitelist.sol (WHL)

An ownable contract meant to sustain a basic whitelist privilege list where the owner of the contract can add and remove members to and from respectively. Additionally, a dedicated getter function exists that allows other contracts as well as the contract itself to validate whether a user is part of the whitelist.

WHL-01M: Inconsistent Whitelist Transitions

Type	Severity	Location
Code Inconsistency	Minor	Whitelist.sol:L20,L25

Description:

The `add` and `remove` functions of the contract do not validate the previous whitelist status of the member being added / removed from the whitelist, thus permitting the `AddedToWhitelist` and `RemovedFromWhitelist` events to be emitted with i.e. members that were never part of the whitelist in the first place.

Recommendation:

We advise a `require` check to be imposed on each function ensuring that the existing status of the member being altered is opposite to the one it is being set to (i.e. `add` should validate that `whitelist[newWhitelisted]` is `false` whilst `remove` should validate that `whitelist[previousWhitelisted]` is `true`).

WHL-01S: Inexistent Visibility Specifier

Type	Severity	Location
Code Style	Informational	Whitelist.sol:L10

Description:

The `whitelist` member has no visibility specifier explicitly set. The current behaviour of the `solc` compiler is to assign a visibility specifier automatically, meaning that an unset visibility specifier can lead to compilation discrepancies should different compiler versions be utilized.

Recommendation:

We advise a visibility specifier to be explicitly set for the variable.

WHL-01C: Constructor Gas Optimization

Type	Severity	Location
Gas Optimization	Informational	Whitelist.sol:L15

Description:

The `constructor` of the contract currently invokes the `add` function internally to add the creator of the contract to the whitelist.

Recommendation:

We advise the `whitelist` entry to be set directly as currently, the `add` function applies the `onlyOwner` modifier redundantly thus incurring an extra gas cost.

Timelock.sol (TML)

A contract meant to represent the internim actuator of a DAO implementation. It is based on the Compound implementation of the same name albeit with a few changes one of which is incorrect and outlined below.

TML-01M: Improper Sanitization of `setPendingAdmin`

Type	Severity	Location
Input Sanitization	Minor	Timelock.sol:L265

Description:

The `setPendingAdmin` function of the contract contains a new check in contrast to the original codebase that validates the `pendingAdmin_` argument is non-zero. This sanitization is incorrect as it prevents un-setting a pending adminsitrator in case one was accidentally or incorrectly set.

Recommendation:

We advise the sanitization to be omitted as it appears to be the result of a misread static analysis finding.